Electronic Version
Stylesheet Version v1.1.1

Description

A code or data representation and manipulation system that allows user to view, modify, and manipulate structured entities, such as code or data files, using their tree-like representations

BACKGROUND OF INVENTION

[0001] Structured entity is the content of a file or memory which follows a set of rules, or messages in a protocol. A set of rules define the format of data inside the structured entity. Examples of structured entities are data files, code files, in-memory structures, files that have formats, and different protocols. All such entities, which are governed by a set of rules, have various applications among information technology areas.

[0002] Data files are used for storing data and exchanging information between applications. File formats define the structure of data files.

[0003] Code files are the files written in different programming or scripting languages. The set of rules, called BNF (Backus-Naur form) or EBNF (Extended BNF) defines the rules for code files. There might be additional rules on top of EBNF as well.

[0004] The structured entities consist of pieces that can be logically or semantically grouped using different criteria. Such pieces will be called information pieces hereinafter.

[0005] There is no unified way of convenient and comprehensible presentation or manipulation of information pieces inside those structured entities. If you open a large file in a regular plain text or binary editor to view or modify the content of the file, it is relatively difficult to navigate to where a specific part of information is stored. The content often contains multiple formatting characters that make the navigation process even harder.

[0006] When the content of a file is modified, there exists the probability of making a mistake or introducing to the file content that does not meet the original structure rules or breaks the file format. Although the changes might look correct, the file becomes "broken", its structure does not follow the rules anymore. A file where the format or rules are broken could easily become inconsistent or not ac-

ceptable by applications that use it. The user may not discover the inconsistencies until the file is submitted to a system that requires a file in a specific format. The system will have to process all the previous records and even perform many other operations before it discovers the error. Such an action could potentially take a significant amount of time depending on input size , system complexity , hardware performance, and other factors.

[0007] Similar problems exist for files with code written using a programming or scripting language. Such files are usually edited within a special environment that gives user some advantages over regular plain text editor such as, for example, syntax coloring and highlighting. However, these environments still do not preserve users from making mistakes. The errors in files written in programming languages will show up only after the compilation or building process, which could take hours, or even days, depending on the size of the project. Another lengthy compilation or building process may be required after the error is discovered and corrected.

[0008] An additional drawback of currently used environments is that usually the user does not have the ability to work on the content of the file in small parts. Modifying a small

part is easier and less error-prone. If users had the ability to modify small parts of the same file separately, more then one user could work on the same file at the same time, which would increase productivity and lead to a finer level of source control.

[0009] U.S. Pat. No. 6,466,240 considers the problems mentioned above and suggests using a data processing system for interactively building a computer program that transforms a tree that represents structured text. However, the patent mentioned above does not provide a complete way of resolving the technical problems. First of all, no method of building such trees is mentioned. The patent describes the manipulations using the tree given that the tree is already built. Meanwhile, building a tree given a structured entity actually plays a key role in the whole method of developing programs in this way. Some of the drawings included in the patent show that nodes of the tree are logically related to a piece of text in the structured text. Nothing is said about the ways of preparing such logical grouping for each program. Secondly, the scope of the claims is restricted to only a few particular types of structured entity and does not handle all types. Thirdly, the system requires the presence of at least two

graphical user interface elements, one of which is the tree and another one is the window that shows the changes in the content according to the changes to the tree.

Summary of Invention

[0010] The present invention aims to provide a way to resolve the problems described above by using specially built tree-like representations. It is suggested that instead of or in addition to regular presentation, every structured entity could be presented as a tree-like view based on the content itself and some additional rules. Tree-like means indentation and the ability to hide subsections of content .

[0011] As opposed to U.S. Pat. No. 6,466,240 , the present invention suggests a revolutionary completely new universal way of building trees for any kind of structured entities, where the the nodes of the tree could be a piece of text in structured text, no other logical grouping is required, and any kind of structured entity can be handled by a system using described method. These trees are not any kind of underlying trees mentioned in the patent described above. They are built based on a unique and completely new algorithm, could be dynamic, and could depend on user preferences or the data inside the structured entity itself.

[0012] Each structured entity type can be described with a set of

rules. For example, BNF/EBNF, which is widely used to define grammars for programming languages, provides a set of such rules. This set of rules can be extended by additional rules, for example, for verification or validation purposes.

[0013] According to the present invention, each rule is coupled with its tree like representation. The tree-like representation for each rule is based on the desired representation pattern for this rule in the main tree-like document that represents an instance of a structured entity and will be constructed later. This coupling can be static (hard-coded) or dynamic (built at run time based on the content of a structured entity or other factors).

[0014] Then when there is a need to build a tree-like representation for an instance of a structured entity (file), those rules are used to map the file content onto those individual trees for each rule and the resulting main tree is built.

[0015] As a result of present invention, structured entities are presented as at least one tree-like structure.

[0016] The tree-like form is easier for reading, editing, and manipulation, all unessential items in the output can be removed or "grayed" and the main aspects can be highlighted and underlined without actual modification of the

original content.

[0017] The result of such representation gives a number of advantages:

[0018] 1. Easier comprehension. More important nodes can be placed at the higher level in the tree or highlighted in another way, and less important nodes can be placed at the lower levels or even hidden. Less important nodes can be shown by drilling down (expanding parent nodes) or by other kinds of additional requests from the user. Thus, a quick look would be enough to find the right information piece inside the structured entity.

[0019] 2. During modification, the structured entity rules can be enforced before the actual modification occurs. The results of node creating, editing, or deleting can be easily verified by the same set of rules and presented as tree-like structures as well. If changes to one or more of the nodes are not correct, the tree-like structure based on the structured entity rules cannot be created, and the modifications won't be accepted. This reduces the number of editing errors to minimum. Many kinds of errors simply cannot be made, as the system won't accept them.

[0020] 3. The trees built in this way can be edited or viewed on the node-by-node basis. Each node or set of nodes can

be presented to the user and processed separately from the main tree-like structure. At the same time it could still act as a part of the main tree-like view or be merged back later. This removes unrelated parts of the structured entity from the user's view and lets the user focus only on the structured entity parts related to the required manipulations. It simplifies working with structured entities, especially with large ones.

[0021] 4. The input verification can become much quicker. There is no need to analyze and process the whole file as it is done in case of compilation and building of code written in a programming language, for example. This process often takes a lot of time, CPU cycles, and other resources. Instead, each node (or a set of nodes) can be considered as a standalone structured entity that must follow a subset or rules, defined for this node/nodes. Each record can be edited in place or using a separate instance of application to isolate it from other records and make sure they are not affected or changed unintentionally. While the record is being changed, changes that do not meet the grammar rule simply cannot be entered, which dramatically reduces the probabilities of any errors. So, only this small piece of input is modified and verified. However, the

quality of verification is still the same. The method utilized by this invention saves time and other resources.

[0022] 5. Tree-like representation makes the process of adding, removing, or editing records much easier. For example, there is no need to scroll through the file to find the right place and copy it, then, find another place, and paste it there. All it takes is selecting a set of nodes in one tree and dragging and dropping it on another or the same tree. The source tree already verified the information pieces that are being transferred. The target tree verifies that the information pieces can be inserted at this location and whether this insertion does not break the rules. The process described is much more advantageous in comparison with simple editing, adding, or deleting pieces of input using regular environments.

[0023] 6. As the representation type can be selected based on preferred criteria, searching becomes much easier as all the unnecessary elements can be either hidden or leveled so that the values by which the search is performed are emphasized.

[0024] 7. Code or data indentation within files is automatically presented by the tree-like structures. The representation can format any structured entity in any usable way ac-

cording to the preferences of the viewer.

Brief Description of Drawings

[0025] Figure 1 shows an example of original representation of a data file in CSV (comma separated values).

[0026] Figure 2 shows an example of a tree-like representation of a file in CSV shown on Figure 1.

[0027] Figure 3 shows an example of another tree-like representation of a file in CSV shown on Figure 1 where the name is emphasized.

[0028] Figure 4 shows an example of original representation of a code file in C Sharp

[0029] Figure 5 shows an example of a tree-like representation of a file in C sharp shown on Figure 4.

[0030] Figure 6 shows an example of original representation of a code file in PHP.

[0031] Figure 7 shows an example of a tree-like representation of a file in PHP shown on Figure 6.

[0032] Figure 8 shows a mapping of one rule for the if statement to a tree like structure

[0033] Figure 9 shows the example of an if statement representation, resulting from the mapping described on Figure 8.

[0034] Figure 10 shows another mapping of the same rule for the if statement to a tree like structure.

[0035] Figure 11 shows the example of an if statement representation, resulting from the mapping described on Figure 10.

[0036] Figure 12 shows an example of editing an if statement separately from the rest of the program.

[0037] Figure 13 shows an example of the original grammar tree for a CSV grammar.

[0038] Figure 14 shows a combined for convenience purposes table illustrating two particular cases of mapping for an if statement in a C++ or C sharp-like programming language.

DETAILED DESCRIPTION

[0039] 1.Structured entities as grammars.

[0040] Structured entities are widely used in information technology areas for different purposes. Structured entities have different formats and follow different rules. Those formats and rules can be potentially described as at least one set of rules in BNF/EBNF format (or similar). A set of such rules is called a grammar. For example, code file formats in programming languages are based on BNF/EBNF grammars.

[0041] 2.Rules in grammars.

[0042] Such grammars consist of rules. Each rule corresponds to a structure that could be encountered inside the structured file. A couple of examples of rules like that is shown below:

[0043] while_statement::= "while" "(" boolean_expression ")" embedded_statement ;

[0044] line ::= value "," line | value "," | value;

[0045] If we take an instance of a structured entity and a related grammar we can construct the grammar tree for this instance. If the structured entity meets all the rules of the grammar, the tree will be complete or otherwise be partial. An example of such grammar tree for an instance of a CSV file and a particular grammar is shown on the Figure 13.

[0046] For more information about grammar trees and how they are constructed please see publicly available books about compilers, shift-reduce parsers, and BNF/EBNF.

[0047] 3.Mapping rules - representations.

[0048] The invention suggests that each rule in the grammar can be mapped to a tree-like representation of the rule elements. The example of such mapping for an IF statement is shown on the Figure 14.

[0049] In the left column of the table you can see two different ways to map the rule to a tree-like structure for the following rule:

[0050] If_statement ::= "if" "(" boolean_expression ")" embedded_statement "else" embedded_statement

[0051] Please refer to BNF/EBNF manuals for more information about BNF/EBNF rules themselves.

[0052] According to the invention, such mappings can be used to build the tree-like representations for each instance of a structured entity.

[0053] 4.An example of a representation building process.

[0054] Based on the original structured entity content or its part a BNF/EBNF tree is built first. Then it is converted to the representation tree using the mappings. An example of this process for a file in C++ or C sharp is shown on the Figure 14. In this example, a code file is considered as a structured entity, and the line

[0055] if (a==1) b=2; else c=3; (1)

[0056] is considered as an information piece. An example of a related rule from the grammar for the files of this type is shown below:

[0057] If_statement ::= "if" "(" boolean_expression ")" embed-

ded_statement "else" embedded_statement

[0058] Note: The rules for Boolean_expression and embedded_statement in the expression above are omitted for clarity

[0059] It can be read as: an if_statement structure should start with an "if", followed by a "(", then a boolean_expression type of structure, then ")", then an embedded_statement type of structure, then "else", then an embedded_statement type of structure.

[0060] The next step is to assign a tree-like representation to rules. Tree like representation is selected based on user needs or other factors, which include but are not restricted to easy and logical presentation and manipulation. The representations could also ba dynamically constructed or depend on the content of the structural entity.

[0061] Two possible representations for the if_statement rule described above are shown on Figure 14 in the left column. They are also shown on Figure 8 and Figure 10. Both add the inner content of the if_statement type of structure as a group of parent-child relationships, so that they can be originally hidden and expanded by a user only if the user is interested to look at them.

[0062] When the original line of code (1) is parsed, the represen-

tation is used to convert this text to a tree-like structure based on the rule representations described above. The examples of final results are shown on Figure 14 in the right column. Note, to construct the tree parts in the right columns the rule for "embedded statement" has been processed in the same way and this process is not described here only for simplicity reasons. The user can now collapse the content of the if-node, which greatly simplifies the comprehension of the code at a higher level. The manipulations are made easier as well because the node can be edited, copied, removed or manipulated in any other way as a whole, with all of its content. If the node is being copied to another tree, for example, the user could be presented with a dialog to edit the content of the node in a tree like or just textual form before pasting the node into the target tree.

[0063] It is to be understood that the if_statement is just an example and the same technology can be applied to any kind of structured entity that can be put in rules.

[0064] 5.Representation types (different mapping of the same inputs).

[0065] Different mappings will result in different tree structures. See the example of two representations of an if_statement

on Figures 8 and 10. The resulting tree structures shown on Figures 9 and 11 are different. One representations shows the "else" part on the same level as the "if" part and another one shows the "else" part as a child of the "if" part which can be more logical in some situations. Figure 14 shows two mentioned above representations and resulting tree-like structures as a table.

[0066] 6. Additional drill down, sub-representations for node of each type.

[0067] It is possible to edit each node of a representation tree by opening it as a separate tree and focusing only on it. Example is shown on Figure 12 where an if statement type is presented as a separate tree. Now it can be modified separately from the main tree and merged with it later. Also additional drilling down is possible for each node based on its meaning and content. If a tree node has it own inner content and grammar, opening it as a separate tree could completely or partially reveal that content in an easy way. It could be shown as a tree-like structure based on its own grammar and rules. For example, it is possible to show machine code or execution time and statistics for each tree node for the case of a programming language based tree.

[0068] 6. Enforcing the rules while changing the content. Comparative analysis of regular compilation vs. suggested tree building.

[0069] Any changes can be parsed back to the tree using the same rules. It means that changes that do not meet the rules will not be possible. For example, if we edit the if_statement separately, as described in 6, and the resulting changes do not meet the rules, it can be determined and reported before the code is merged back to the main tree. Only the trees that meet the rules can be merged with the main tree, and only the changes that meet the rules can be made, which greatly reduces the number of errors and time, spent by compiler at build/compilation time to check that the code follows the rules.

[0070] 7. Building a representation for a binary file.

[0071] Binary files can also be presented as a tree like representation. For example, a picture in BMP format can be presented as a tree, where nodes are image sizes, creator, dates, format identifiers, and a set of lines of the image. In that way binary resources can be easily manipulated on the low level and are just a particular case of a structured entity, which means that all that is described in this invention is equally applicable to them too.

[0072] 8. Code copying examples.

[0073] The code or data presented in this tree-like way can be easily copied/replicated/manipulated in logically grouped pieces. For example, you could copy the entire content of an if_statement, just as you copy a folder in windows explorer. Multiple tree nodes can also be involved in the operation at the same time. This simplifies the code management and makes code composition easier. It is also profitable to implement source control based on such nodes and not on the whole file. In this way, multiple people could lock and manipulate different nodes in the same file at the same time.

[0074] The invention being thus described, it will be evident that the same may be varied in many ways. Such variations are not to be regarded as a departure from the spirit and scope of the invention and all such modifications are intended to be included within the scope of the claims.